

8 Loops

8.0	Introduction
8.1	Introduction to Loops
8.2	Pseudo-code
8.3	Counted Loops
8.4	Conditional Loops
8.5	Choosing Which Loop
8.6	Breaking Out of Loops
8.7	Nesting Loops
8.8	Technical Terms
8.9	What You Should Know
8.10	Quiz Answers
8.11	Exercises

8.0 Introduction

This chapter covers the many aspects of the loop construct including the two different types of loops, exiting loops, and nesting loops (placing a loop inside another loop). It also introduces the concepts of pseudo-code, a program-like notation that can be used to design algorithms without having to handle the details of a particular programming language.

8.1 Introduction to Loops

In all the programs we have seen so far, a new instruction has been issued for every action that the program is to take. This is known as **sequential execution** and is enormously inefficient in tasks where there is a great deal of repetition. For example, in the *TermDeposit* program the statements to calculate and output the interest were repeated five times (once for each year).

To avoid this inefficiency and allow greater flexibility, programming languages have a construct called a loop. A **loop** is a construct that allows a program to execute a given set of instructions either a specified number of times or while a condition is true.

A loop that executes a specified number of times is called a **counted loop**. A loop that executes while a condition is true is called a **conditional loop**. Here are two real life examples of these two types of loops.

Counted Loop – Driving instructions for someone who hates making left turns at crowded intersections:

- *Execute the following three times*
 - *Go forward one block*
 - *Turn right*

Conditional Loop – Instructions for issuing refunds for a cancelled rock concert.

- *Execute the following while people are waiting in line*
 - *Get the ticket number of the first person in line*
 - *Verify a refund has not been issued for this ticket*
 - *Take ticket from person*
 - *Give refund to person*

Of course, sequential execution and loops can be combined. Here is an example of instructions for channel surfing.

- *Find remote control*
- *Sit down on couch*
- *Turn on TV with remote control*
- *Execute the following while the program on TV is boring*
 - *Click the “Channel+” button*
 - *Wait until commercials are finished*
- *Relax and watch rest of show*

8.2 Pseudo-code

An algorithm is a specification of a solution for a problem. However, the specification can be made in a variety of ways. As we have seen, using pure English is not ideal for specifying instructions because English is ambiguous. While computer languages are ideal for eliminating ambiguity, the complexity of the syntax required in a computer language can often obscure the understanding of the algorithm. As well, expressing an algorithm in a specific programming language means that those who are not familiar with that particular language cannot easily use the algorithm.

A mid-way solution is the use of pseudo-code. **Pseudo-code** is a notation for expressing algorithms that combines the basic constructs of programming with informal nature of English. It combines the accuracy of a programming language without its extra syntax. In general, the reader can resolve any ambiguity in a pseudo-code specification.

Here is the pseudo-code equivalent of the *TermDeposit* program in Section 5.3.2.

- Obtain the principal and interest rate as a percent from the user.
- Convert the interest rate as a percent into a decimal number
- For each of five years
 - `interest = principal * interest rate`
 - `interest = ((int) (interest * 100)) / 100`
 - `principal = principal + interest`
 - Output year, interest, and principal

In this pseudo-code example, some elements are expressed exactly (such as the rounding of the interest to the nearest penny), while other elements are only sketched out (such as obtaining numbers from the user and the format of the output). The creator of the pseudo-code must decide which elements of the algorithm are self-evident to the reader and which must be explained in greater detail.

The informal nature of pseudo-code means that there is no absolute standard. Some write pseudo-code that is quite close to actual program code while others write pseudo-code that is more informal. The success of pseudo-code rests in how effectively it communicates the non-obvious elements of the algorithm it expresses.

8.3 Counted Loops

A counted loop is a loop that repeatedly executes a set of statements a specified number of times. However, you do not specify the number of times the loop is to execute directly. Instead, you create a variable (called the **index variable**), assign it an **initial value**, and then execute the statements of the loop until the index value reaches a **termination value**.

8.3.1 Simple Counted Loops

Here is the simplified syntax for a counted loop in Java.

```
for (int index-variable = initial-value ;
     index-variable < termination-value ;
     index-variable++)
{
    ... loop statements ...
} // for
```

When execution reaches the `for` loop, the index variable is created and assigned the initial value. The index variable is then compared with the termination value. If the index variable is less than the termination value, then the statements in the loop (commonly called the **loop statements**) are executed.

After the loop statements are executed, execution returns to the `for` statement. The index variable is incremented by one and compared with the termination value.

If the index value is less than or equal to the termination value, the loop statements are executed once again and execution returns to the `for` loop for another increment and comparison. Otherwise, execution continues at the statement following the loop statements.

Each execution of the loop statements is called an **iteration**. Thus, we speak of two iterations of a loop, meaning that the loop statements are executed twice. We also say the execution **iterates**, meaning it repeatedly executes the loop statements. If we speak about a loop that **iterates over** or **iterates through** a data structure, it means that the loop statements execute once for each element of the data structure.

For Loop Example

Here is an example of a loop in Java that outputs the numbers 0, 1, and 2 on separate lines.

```
for (int i = 0 ; i < 3 ; i++)
{
    System.out.println (i);
} // for
```

In this example, the index variable is `i`, the initial value is 0, the termination value is 3, and there is only one statement to be executed repeatedly: `System.out.println`.

Here is the trace of the execution of the `for` loop.

When the `for` statement is first executed, the index variable (`i`) is declared as an integer and assigned the initial value (0). The index variable is compared with the termination value of 3 and the statement in the loop (the one in curly braces) is executed once. In this case, it outputs 0 to the system console.

After concluding the execution of the loop statements, execution returns to the `for` statement. The index variable is incremented, and the `i` is set to 1. Then the index variable is compared with the termination value. Because `i` is less than 3, the loop statements are executed again.

This time, the loop statements output 1 to the system console. Execution returns once again to the `for` statement. The index variable `i` is incremented to 2 and compared with 3 and the loop statements are executed again. The loop statements output 2 to the system console.

Execution once again returns to the `for` loop. The index variable `i` is incremented to 3 is again compared with 3. This time, the comparison evaluates to `false`. Execution jumps to the statement after the loop statements.

Important Points About For Loops

There are some important points to know about loops.

- The three sections of the `for` statement (initialization section, comparison section, and increment section) are separated by semicolons. There is no semicolon after the increment section.
- There is no semicolon after the `for` statement! This is a very common error and the Java compiler does not detect it. The effect of placing a semicolon after the `for` statement is to not execute the loop statements at all until execution leaves the statement. At that point, the intended loop statements are executed exactly once.
- The index variable is declared in the `for` statement. It exists in the loop statements and ceases to exist after the loop statements. (i.e. the scope of the index variable is the loop statements.) It is an error to declare an index variable with the same name as an already existing variable in your program.
- The first comparison in the `for` statement takes place before the loop statements are executed even once. The statements

```
for (int counter = 10 ; counter < 3 ; counter++)  
{  
    System.out.println ("Hello");  
} // for
```

output nothing. The statement in the loop is skipped because the initial value of counter (10) is greater than or equal to the termination value (3).

- While it is not an error in Java, it is very bad programming practice to change the value of the index variable in the loop statements. It should be obvious looking at the `for` statement how many time the loop will execute. Changing the index variable violates this understanding by the reader.
- The comparison does not have to be a simple less than (<). If it is appropriate, a less than or equals to (<=) comparison is also acceptable.
- It is a good idea to put a one-line comment at the end of the closing brace of the loop statements. While it is obvious in short loops which closing brace is related to which statement, as there are more loop statements, the comment reduces reader confusion.

Ready Tips

If you use the indenter in Ready, a semicolon after the `for` statement will be moved to the next line and indented, which will bring the unintended semicolon to your attention. Here is an example of how a `for` statement with a semicolon would be indented.

```
for (int i = 0 ; i < 3 ; i++)  
    ;  
{  
    System.out.println (i);  
} // for
```

A More Complicated For Loop Example

Here is a slightly more complicated example from the *SquareSquareRoot* program that outputs a table of numbers from 1 to 10 along with their square and their square root.

```
// From main method of the "SquareSquareRoot" class.
for (int number = 1 ; number <= 10 ; number++)
{
    Stdout.print (number, 2);
    Stdout.print (number * number, 5);
    Stdout.println (Math.sqrt (number), 8, 4);
} // for
```

In this example, the index variable *number* is declared and initialized to 1. The loop statements are executed and output the line

```
1    1  1.0000
```

Once the loop statements are executed, execution returns back to the **for** statement. The index variable *number* is incremented to 2 and the comparison is made. This comparison will evaluate to **true** while *number* is less than or equal to 10. This means the loop statements will execute nine more times producing:

```
2    4  1.4142
3    9  1.7321
4   16  2.0000
5   25  2.2361
6   36  2.4495
7   49  2.6458
8   64  2.8284
9   81  3.0000
10  100 3.1623
```

The final time, *number* will be incremented to 11. The comparison will evaluate to **false** and execution will pass to the statements following the loop statements.

The *TermDeposit* Program Revisited

In the original *TermDeposit* program, the interest was calculated on a term deposit over 5 years. This required repeating the interest and output statements five times. By using a counted loop, the code to calculate the interest and output the data occurs only once inside a loop.

Using a **for** loop makes the program smaller, and just as importantly, easier to read and understand. It also makes the program much more flexible. By changing one line in the program, it is possible to make the program calculate the interest over 10 years.

Here is the *TermDeposit4* program.

```
// The "TermDeposit4" class.
import java.text.NumberFormat;
import hsa.Stdin;
import hsa.Stdout;

public class TermDeposit4
{
    public static void main (String[] args)
    {
        // The current balance of the term deposit.
        double amount;
        // The interest rate that the term deposit earns
        // each year, in percent.
        double interestRateInPercent;
        // The interest earned in a given year.
        double interestPaid;

        double interestRateAsNumber;
        int interestInPennies;

        // Formatter of currency.
        NumberFormat currencyFormatter;

        System.out.println ("Enter the initial principal and");
        System.out.println ("interest for a 5 year term deposit");

        System.out.print ("Enter principal for term deposit: ");
        amount = Stdin.readDouble ();
        System.out.print ("Enter the interest rate in percent: ");
        interestRateInPercent = Stdin.readDouble ();

        interestRateAsNumber = interestRateInPercent / 100;

        // Obtain the currency formatter.
        currencyFormatter = NumberFormat.getCurrencyInstance ();

        for (int year = 1 ; year <= 5 ; year++)
        {
            interestPaid = amount * interestRateAsNumber;
            interestInPennies = (int) (interestPaid * 100);
            interestPaid = interestInPennies / 100.0;

            amount = amount + interestPaid;

            Stdout.print (year, 2);
            Stdout.print (" ");
            Stdout.print (currencyFormatter.format (interestPaid),
                - 10);
            Stdout.print (" ");
            Stdout.println (currencyFormatter.format (amount), -10);
        } // for
    } // main method
} // TermDeposit4 class
```

8.3.2 Quiz Yourself – Simple Counted Loops

Answer the following questions. See the page 311 for worked out answers.

What is the output for the each group of statements? If the program would not compile, then indicate that instead of the output.

- a.

```
for (int j = 0 ; j < 4 ; j++)
{
    System.out.print ("Hi!");
} // for
```
- b.

```
for (int j = 0 ; j < 4 ; j++)
{
    System.out.println (5 - j);
} // for
```
- c.

```
for (int j = 1000 ; j <= 1005 ; j++)
{
    System.out.println (5 - j);
} // for
```
- d.

```
for (int j = 10 ; j < 12 ; j++);
{
    System.out.println ("Hello");
} // for
```
- e.

```
for (int j = 6 ; j < 6 ; j++)
{
    System.out.println (j);
} // for
```
- f.

```
for (int j = 10 ; j <= 10 ; j++)
{
    System.out.println (j);
} // for
```
- g.

```
for (int j = -6 ; j < -4 ; j++)
{
    System.out.println (5 - j);
} // for
```

8.3.3 More Complicated Counted Loops

In the previous section, we assumed that the counted loop would always increment the index variable by one. Counted loops in Java, however, are somewhat more flexible. Counted loops can count downwards as well as upward and they can increment or decrement by values other than one. As well, the index variable can be compared using any logical expression.

Here is the syntax for a counted loop in Java.

```

for (initialization-section ;
      comparison-section ;
      increment-decrement-section)
{
    ... loop statements ...
} // for

```

Counting Backwards

To make a loop that counts down, the index value is declared and initialized as usual. However, in a loop that counts down, the comparison should check that the index variable is greater than (or greater than or equal to) the termination value. Finally, the increment/decrement section should decrease the value of the index variable. The `--` operator is often used to decrement a variable by one.

Here is a `for` loop that counts backwards from 5 to 1

```

for (int i = 5 ; i >= 1 ; i--)
{
    System.out.println (i);
}

```

Here is the trace of the execution of the `for` loop. The line being executed is indicated with an asterisk (*).

<pre> * for (int <i>i</i> = 5 ; <i>i</i> >= 1 ; <i>i</i>--) { System.out.println (<i>i</i>); } </pre>	The index variable (<i>i</i>) is declared and assigned the initial value (5). The comparison evaluates to <code>true</code> . The loop statements are executed.	<i>i</i> = 5
<pre> for (int <i>i</i> = 5 ; <i>i</i> >= 1 ; <i>i</i>--) { * System.out.println (<i>i</i>); } </pre>	Output 5 on the system console.	
<pre> * for (int <i>i</i> = 5 ; <i>i</i> >= 1 ; <i>i</i>--) { System.out.println (<i>i</i>); } </pre>	The index variable <i>i</i> is decremented to 4. The comparison evaluates to <code>true</code> . The loop statements are executed.	<i>i</i> = 4
<pre> for (int <i>i</i> = 5 ; <i>i</i> >= 1 ; <i>i</i>--) { * System.out.println (<i>i</i>); } </pre>	Output 4 on the system console.	
<pre> * for (int <i>i</i> = 5 ; <i>i</i> >= 1 ; <i>i</i>--) { System.out.println (<i>i</i>); } </pre>	The index variable <i>i</i> is decremented to 3. The comparison evaluates to <code>true</code> . The loop statements are executed.	<i>i</i> = 3
<pre> for (int <i>i</i> = 5 ; <i>i</i> >= 1 ; <i>i</i>--) { * System.out.println (<i>i</i>); } </pre>	Output 3 on the system console.	

<pre>* for (int i = 5 ; i >= 1 ; i--) { System.out.println (i); }</pre>	The index variable <i>i</i> is decremented to 2. The comparison evaluates to true . The loop statements are executed.	<i>i</i> = 2
<pre>for (int i = 5 ; i >= 1 ; i--) { System.out.println (i); }</pre>	Output 2 on the system console.	
<pre>* for (int i = 5 ; i >= 1 ; i--) { System.out.println (i); }</pre>	The index variable <i>i</i> is decremented to 1. The comparison evaluates to true . The loop statements are executed.	<i>i</i> = 1
<pre>for (int i = 5 ; i >= 1 ; i--) { System.out.println (i); }</pre>	Output 1 on the system console.	
<pre>* for (int i = 5 ; i >= 1 ; i--) { System.out.println (i); }</pre>	The index variable <i>i</i> is decremented to 0. The comparison evaluates to false . Execution continues following the loop statements.	<i>i</i> = 0

Here is another **for** loop that outputs 8 to -8 on one line on the system console using the *Stdout* class.

```
for (int count = 8 ; count >= -8 ; count--)
{
    Stdout.print (count, 3);
} // for
```

This loop outputs

```
8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5 -6 -7 -8
```

Matching Comparison with Increment/Decrement

You must be very careful when creating loops, especially those that count down. If the initial value, the comparison, and the increment do not match, it is easy to write a **for** loop in which the loop statements are either never executed, or continuously executed until the program is halted by the user.

Here is an example of a **for** loop where the loop statements are never executed.

```
for (int year = 2000 ; year > 2005 ; year++)
{
    System.out.println ("Welcome to " + year);
} // for
```

In this example, the initial comparison evaluates to **false** (2000 is not greater than 2005) and execution continues after the loop statements without ever executing them.

Here is a second example.

```
for (int counter = 10 ; counter >= 1 ; counter++)
{
    System.out.println (counter);
} // for
```

In this example, *counter* is initialized to 10. The comparison evaluates to **true** and after executing the loop statements, the counter is incremented by one. Unfortunately, this means that the comparison will never evaluate to **false** and the loop statements will execute indefinitely until the user halts the program. The output is

```
10
11
12
13
...on forever...
```

A loop that, intentionally or not, never exits is called an **infinite loop**. An unintentional infinite loop usually results from a logic error in a program and forces the user to terminate the program.

Ready Tips

You can always terminate execution of a program in the environment by clicking on the Close button in the Standard Input/Output window.

Loops that Count Up or Down by More than One

The **for** loop in Java is not constrained to count by ones. The increment section can change the value of the index variable by any amount. Here is a **for** loop that counts from 7 to 49 by 7's.

```
for (int count = 7 ; count <= 49 ; count += 7)
{
    System.out.println (count);
} // for
```

In this example, the **for** loop operates as before, except that after each execution of the loop statements, the index variable *count* is incremented by 7 rather than 1. The increment section in this example uses the += assignment operator. It could have also used the statement

```
count = count + 7
```

Note that in either case, there is no semicolon after the increment section.

Here is an example of the loop that outputs the value of the sine function for angles from 0 to 360 every 10 degrees. The first statement in the loop converts the angle in degrees into an angle in radians. The second outputs the sine to 4 decimal places.

```
// From main method of the "Sines" class.
for (int angle = 0 ; angle <= 360 ; angle += 10)
{
    double angleInRadians = Math.toRadians (angle);
    Stdout.println (Math.sin (angleInRadians), 7, 4);
} // for
```

The first time through the loop *angle* has a value of 0, the second time through the loop *angle* has a value 10, then 20, 30, and so on. The program finishes when *angle* is incremented from 360 to 370 and the comparison condition evaluates to **false**.

Java allows you to have any expression that evaluates to a **boolean** value as the comparison condition. Likewise, the increment section does not have to modify the index variable. However, this can confuse readers because it violates standard practices for use of a **for** statement and thus should be avoided.

8.3.4 Quiz Yourself – Counted Loops

Answer the following questions. See the page 313 for worked out answers.

What is the output for the each group of statements? If the program would not compile, then indicate that instead of the output. If the program would produce infinite output, write the output for the first few lines and then a “. . .”

- a.

```
for (int j = 6 ; j > 2 ; j--)
{
    System.out.print ("Hi!");
} // for
```
- b.

```
for (int j = 0 ; j < 4 ; j += 2)
{
    System.out.println (5 - j);
} // for
```
- c.

```
for (int j = 12 ; j <= 20 ; j--)
{
    System.out.println (j);
} // for
```
- d.

```
for (int j = -5 ; j < 5 ; j++)
{
    System.out.println (j);
} // for
```
- e.

```
for (int j = -3 ; j >= -21 ; j -= 3)
{
    System.out.println (j);
} // for
```

```
f. for (int j = 10 ; j <= 3 ; j--)
    {
        System.out.println (j - 5);
    } // for
g. for (int j = -10 ; j < 10 ; j += 30)
    {
        System.out.println (j);
    } // for
```

8.4 Conditional Loops

A conditional loop is a loop that executes while a certain Boolean condition holds true. There are two forms of conditional loops. The **while** loop evaluates a Boolean expression and then executes a set of statements (the loop statements) if the expression is **true** before evaluating the expression again. The **do** loop executes the loop statements, and then evaluates a Boolean expression. If the expression evaluates to **true**, then it executes the statements again before repeating the evaluation.

8.4.1 While Loops

Here is the syntax for a **while** loop in Java.

```
while (boolean-expression)
{
    ... loop statements ...
} // while
```

When execution reaches the **while** statement, the Boolean expression is evaluated. If the result is **true**, then the loop statements are executed and execution then returns to the **while** statement. The expression is again evaluated and the loop statements executed again if the result is **true**. The evaluation and execution of the loop statements continues until the expression evaluates to **false**, whereupon execution continues at the statement after the closing brace.

While Loop Example

Here is a loop that sums a bunch of numbers until the user enters **-1**, whereupon it outputs the sum on the system console.

```
// From main method of the "CalcSum" class.
int number, sum = 0;
number = Stdin.readInt ();
while (number != -1)
{
    sum = sum + number;
    number = Stdin.readInt ();
} // while
System.out.println ("The sum is " + sum);
```

In this example, the boolean condition is `number != -1` (`number` does not equal `-1`). The code starts by declaring `number` and `sum` and initializing `sum` to 0. An integer is then read in from the user and assigned to `number`. Execution reaches the `while` statement. The expression is evaluated. If the user entered `-1`, the expression evaluates to false and execution continues at the `System.out.println` statement. Otherwise, the expression evaluates to `true` and the loop statements are executed. First `sum` is incremented by `number`. Then a new integer is read in from the user. At this point the end of the loop is reached and execution returns to the `while` statement.

The Boolean expression is evaluated again and if the `number` is not `-1`, then the loop statements are executed again. The loop statements will continue to be executed until the user enters `-1`.

The `-1` in this example is often called a sentinel value. A **sentinel value** is a special value to be entered by the user that marks the end of input. A sentinel value can be a number or a string.

Important Points About While Loops

Do not place a semicolon after a `while` statement. Like the `for` statement, it is legal to put a semicolon after a `while` statement and will not cause a compilation error. However, it will likely cause an infinite loop. A semicolon by itself represents the **null statement**, a statement that does nothing. Instead of executing the loop statements after the condition has been evaluated to true, it executes the null statement. It then returns to the `while` statement. Since no variables have changed, the condition will still be true and the program will loop executing the null statement and checking the condition until the user halts the program.

Ready Tips

As with the `for` loop, using the indenter will make it obvious if you have accidentally added a semicolon after a while loop.

```
while (sum < 10)
    ;
{
    sum = sum + value;
} // while
```

It is good programming practice to add a one line comment (`//` style comment) to the closing brace of the while loop indicating the closing brace is associated with the while loop. This helps readers follow your program more easily.

A More Complicated While Loop Example

In this example, a program asks a user for a list of positive numbers, terminated with `-1`. It then outputs the mean (average) of the numbers entered.

Here is the pseudo-code for a program to calculate the averages.

- Set the sum and number-of-entries to be 0.
- Read a number from the user
- while the number is not -1
 - sum = sum + number
 - number-of-entries++
 - Read a number from the user
- Output the sum / number-of-entries

The pseudo-code represents the algorithm of the program. Here is the *CalcAverage* program represented by the pseudo-code.

```
// The "CalcAverage" class.
import hsa.Stdin;

public class CalcAverage
{
    public static void main (String[] args)
    {
        int sum = 0, numberOfEntries = 0;
        int number = Stdin.readInt ();
        while (number != -1)
        {
            sum += number;
            numberOfEntries++;
            number = Stdin.readInt ();
        } // while
        System.out.print ("The average is ");
        System.out.println ((double) sum / numberOfEntries);
    } // main method
} // CalcAverage class
```

Note the necessity of casting *sum* to a **double** before performing the division. Without the cast, the program would use integer division, truncating the result.

8.4.2 Quiz Yourself – While Loops

Answer the following questions. See the page 315 for worked out answers.

What is the output for the each group of statements? If the program would not compile, then indicate that instead of the output. If the program would produce infinite output, write the output for the first few lines and then a “...”

- a.

```
int j = 1
while (j <= 8)
{
    System.out.print (j);
    j = j * 2;
} // while
```

- b. `int j = 20`
`while (j > 3)`
`{`
`System.out.println (j);`
`j = j / 2;`
`} // while`
- c. `int number = 1, sum = 20;`
`while (number < 5);`
`{`
`sum += number;`
`number++;`
`System.out.println (sum);`
`} // while`
- d. `int number = 1, product = 20;`
`while (product < 100)`
`{`
`product *= number;`
`number++;`
`System.out.println (number);`
`} // while`
- e. `int number = 3;`
`while (number < 20)`
`{`
`number = number + (number - 2);`
`System.out.println (number);`
`} // while`
- f. `int number = 5`
`while (Math.pow (number, 2) < 50)`
`{`
`System.out.println (number);`
`number++;`
`} // while`
- g. `int j = -10`
`while (!(j > 10))`
`{`
`System.out.println (j);`
`j += 3;`
`} // while`

8.4.3 Do Loops

Here is the syntax for a `do` loop in Java.

```
do
{
    ... loop statements ...
}
while (boolean-expression);
```

A **do** loop starts with the keyword **do**, is followed by the loop statement in braces and ends with the boolean condition in a **while** statement. In a **do** loop, the **while** statement must be terminated with a semicolon.

In a **do** loop, the loop statements are executed first, and then the Boolean expression in the **while** statement is evaluated. If the result is **true**, then the loop statements are executed again and execution then returns to the **while** statement. The expression is again evaluated and the loop statements executed again if the result is **true**. The evaluation and execution of the loop statements continues until the expression evaluates to **false**, whereupon execution continues at the statement after **while** statement.

In short, a **do** loop is like a **while** loop except that the loop statements are executed once before the Boolean expression is first evaluated. A **do** loop is often used instead of a **while** loop when the calculations needed to evaluate the Boolean condition are performed in the loop itself.

Do Loop Example

Here is a loop keeps asking for input from the user until they enter an even number.

```
// From main method of the "EvenNumber" program.
int value;
do
{
    System.out.print ("Enter an even number: ");
    value = Stdin.readInt ();
}
while ((value % 2) != 0);
```

In this example, the loop statements output a prompt and read an integer from the user. The integer is then tested to see if it is even. (A number is even if the number modulo 2 is 0.) If *value* is odd then the Boolean expression evaluates to **true** and the loop statements are executed again. If *value* is even, then the expression is **false** and execution continues on the line following the **while** statement.

The example used a **do** loop instead of a **while** loop because the calculations necessary to evaluate the Boolean condition (i.e. obtaining reading *value* from the user) were performed inside the loop.

Important Points About Do Loops

The most important point is that in the **do** loop, the **while** statement has a semicolon after it, while in a **while** loop, the **while** statement does not have a semicolon after it.

Because the braces that enclose the loop statement in a **do** loop are surrounded by the **do** and **while** statements, it is not necessary to place a comment on the closing brace. Readers seeing the closing brace followed by a **while** statement with a semicolon will know that it is part of the **do** loop.

A More Complicated Do Loop Example

Here is an example of a `do` loop in a program using the `PaintBug` object. In this example, the bug moves in a random walk. The bug takes steps in a random direction, eventually reaching the edge of the screen. When it reaches the edge of the screen, the program halts.

The pseudo-code for the program is as follows

- create the `PaintBug` object
- loop
 - calculate a random angle from 0-360
 - set the bug's direction to the angle
 - move 20 units in the current direction
 - get the bug's current x,y position
 - repeat loop if x,y in the window

The program uses a `do` loop instead of a `while` loop because the calculations that are used to decide whether to continue execution of the loop statements are performed in the loop itself. In other words, it is not possible to evaluate the Boolean condition until execution has passed through the loop at least once. In situations like this, it usually makes sense to use a `do` loop.

Note that the position of the bug is returned as floating-point numbers, while the window's width and height are integers. There is no need to perform a cast in order to compare floating-point numbers and integers.

Here is the `RandomWalk` program.

```
// The "RandomWalk" class.
import hsa.PaintBug;
public class RandomWalk
{
    public static void main (String[] args)
    {
        PaintBug bug = new PaintBug ();
        int width = PaintBug.getXSize ();
        int height = PaintBug.getYSize ();
        double x, y;

        do
        {
            double angle = Math.random () * 360;
            bug.setDirection (angle);
            bug.move (20);
            x = bug.getXPos ();
            y = bug.getYPos ();
        }
        while ((0 <= x) && (x < width) &&
              (0 <= y) && (y < height));
    } // main method
} // RandomWalk class
```

8.5 Choosing Which Loop

When designing an algorithm using loops, it is important to choose the correct loop. The different types of loops in Java are flexible enough that it is possible in general to use each type of loop to emulate the others. However, it makes it much harder for programmers reading the program to understand the program's flow.

8.5.1 Counted Loops vs. Conditional Loops

The first decision in choosing which type of loop to use in the implementation of an algorithm is to decide whether to use a counted loop or a conditional loop. Here are some guidelines as to which type of loop to use.

- Counted loops should only be used when the loop is to be executed a specific number of times.
- Conditional loops are used when looping should continue until certain conditions are met.
- A loop that is supposed to be executed either a certain number of times or until a condition is met should usually be implemented as a conditional loop.
- Likewise, if you find that the most natural implementation of your algorithm involves modifying the index variable of a counted loop in the loop statements, you should use a conditional loop instead.

Examples of Choosing the Type of Loop

Here are examples of choosing the type of loop to be used in given circumstances.

- **A loop that counts months from 1 to 12**
This loop always executes 12 times. Use a counted loop.
- **A loop that counts from the year 1996 to the year 2006.**
This loop always executes a set number of times. Use a counted loop.
- **A loop that executes moves a PaintBug until hits the edge of the window.**
This loop should be implemented as a conditional loop.
- **A loop that executes for 5 years or until a mortgage is paid off.**
This loop normally executes 5 times, but it might occasionally execute fewer times. Since the loop will usually execute fully, a counted loop could be used, although a conditional loop is also acceptable.

While Loop vs. Do Loop

When using a conditional loop, the second decision is whether to use a `while` loop or a `do` loop. Here are some guidelines as to which type of conditional loop to use.

- If there are situations that the loop statements in a conditional loop should not execute at all, use a **while** loop instead of a **do** loop.
- If the loop must execute at least once before a comparison is made, use a **do** loop instead of a **while** loop.
- If the loop could be equally easily implemented as either a **while** or a **do** loop, use a **while** loop. The Boolean condition at the top of the loop makes it easier for readers of the program to determine when the purpose of the loop.
- If many calculations needed to evaluate the Boolean condition, consider a **do** loop in order to avoid performing the calculations both before the loop and in the loop.

8.6 Breaking Out of Loops

Here is a simple algorithm for calculating the mean of an unknown number of values.

- `number-of-values = 0, sum = 0`
- `loop`
 - `get a value from the user`
 - `exit the loop if the value is -1`
 - `sum += value`
 - `number-of-values++`
- `mean = sum / number-of-values`
- `Output mean and number-of-values`

Unfortunately, this algorithm tries to exit a loop from the middle of the loop, rather than from the top (as in a **while** loop) or the bottom (as in a **do** loop). In order to make the algorithm conform to a loop that always exits at either the top or at the bottom, the algorithm must be rewritten as

- `number-of-values = 0, sum = 0`
- `get a value from the user`
- `loop while value is not -1`
 - `sum += value`
 - `number-of-values++`
 - `get a value from the user`
- `mean = sum / number-of-values`
- `Output mean and number-of-values`

This means that we have had to copy a part of the loop

- `get a value from the user`

before the loop and then rearrange the internals of the loop. While this is not too difficult, our original pseudo-code was easier to understand and thus less prone to mistakes.

Fortunately, Java has a construct to allow us to exit the middle of a loop.

8.6.1 The Break Statement

The **break** statement is used to exit from loops (as well as from **switch** statements). The syntax is simply

```
break;
```

When the **break** statement is executed, execution continues immediately after the enclosing loop.

The **break** statement is usually used with an **if** statement. When the Boolean condition in the **if** statement is true, then the **break** statement is executed and execution leaves the loop.

```
if ( ...condition... )
{
    break;
} // if
```

Example of Break Statements

You can use the break statement to create loops that more accurately follow the structure of your algorithms. Here is an example of the first algorithm implemented in Java.

```
// From main method of the "ReadAndSum" class.
int numberOfValues = 0, sum = 0;
while (true)
{
    int value = Stdin.readInt ();
    if (value == -1)
    {
        break;
    }
    sum += value;
    numberOfValues++;
} // while
Stdout.print ("The mean is ");
Stdout.println ((double) sum / numberOfValues, 0, 2);
```

The statement

```
while (true)
```

is an infinite loop. The Boolean expression in the parentheses always evaluates to **true**, so execution continues until a **break** statement is executed. In this case, that occurs when the user enters `-1` at the keyboard.

Another Example of Break Statements

A common use for exiting in the middle of a loop is when validating user input. The pseudo-code for such a loop is

- loop
 - prompt user for input
 - get input from user
 - exit loop if input is valid
 - output an error message

Here is a small Java program that asks the user for a value from 1 to 10.

```
// From main method of the "OneToTen" class.
int number;
while (true)
{
    System.out.print ("Enter a number from 1-10: ");
    number = Stdin.readInt ();
    if ((1 <= number) && (number <= 10))
    {
        break;
    } // if
    System.out.println ("Bad Input! The number must " +
        "be between 1 and 10.");
} // while
```

8.6.2 Break Statements and Counted Loops

While it is permissible to have a **break** statement in a counted loop, the practice of doing so should be avoided. In general, a reader encountering a **for** loop expects that the loop will be executed the number of times specified by the initial value, the termination value, and the increment section.

Placing **break** statements in the **for** loop makes this expectation fail. This failure of expectations make a program harder to read and easier to misunderstand. Thus, if a loop requires **break** statements, it is better to implement the loop as conditional loop rather than a counted loop.

Implementing a **for** loop using a **while** loop is straightforward. The following **for** loop

```
for (int index-var = initial-value ;
    comparison-condition ; increment-section)
{
    ...loop statements...
} // for
```

can be implemented using the following **while** loop.

```
index-var = initial-value;
while (comparison-condition)
{
    ...loop statements...
    increment-section
} // while
```

8.6.3 Multiple Break Statements

A loop may have multiple break statements within it. When execution reaches any **break** statement, then execution continues after the end of the loop. Note that it is perfectly allowable to use both a Boolean expression in a **while** or **do** loop that uses **break** statements. However, remember that the Boolean expression in the **if** statement that surrounds the **break** statement should evaluate to **true** if execution should leave the loop and the Boolean expression in the **while** statement should evaluate to **false** if execution should leave the loop.

Here is a longer program that places two *PaintBug* objects in the window. The two then stagger about in a random walk until either one of them reaches the edge of the screen, or they come within 50 pixels of each other.

Here is the pseudo-code for the program.

- create two *PaintBugs*
- loop
 - set the first bug's direction to a random angle from 0-360
 - set the second bug's direction to a random angle from 0-360
 - move the first bug 20 units in its current direction
 - get the first bug's current x,y position
 - if the bug is off the window, exit the loop
 - move the second bug 20 units in its current direction
 - get the second bug's current x,y position
 - if the bug is off the window, exit the loop
 - calculate the distance between the two bugs
 - continue looping if the distance is greater than 50.

Here is the *RandomTwoWalk* program developed from the pseudo-code.

```
// The "RandomTwoWalk" class.
import java.awt.Color;
import hsa.PaintBug;

public class RandomTwoWalk
{
    public static void main (String[] args)
    {
        int width = PaintBug.getXSize ();
        int height = PaintBug.getYSize ();
        double x1, y1, x2, y2, distance;
        PaintBug bug1 = new PaintBug (width / 3, height / 2);
        PaintBug bug2 = new PaintBug (2 * width / 3, height / 2);

        do
        {
            bug1.setDirection (Math.random () * 360);
            bug2.setDirection (Math.random () * 360);
```

```

        bug1.move (20);
        x1 = bug1.getXPos ();
        y1 = bug1.getYPos ();
        if ((x1 < 0) || (x1 >= width) ||
            (y1 < 0) || (height <= y1))
        {
            break;
        }

        bug2.move (20);
        x2 = bug2.getXPos ();
        y2 = bug2.getYPos ();
        if ((x2 < 0) || (x2 >= width) ||
            (y2 < 0) || (height <= y2))
        {
            break;
        }

        distance = Math.sqrt (Math.pow ((x1 - x2), 2) +
                                Math.pow ((y1 - y2), 2));
    }
    while (distance > 50);
} // main method
} // RandomTwoWalk class

```

The first thing to note is that there is not an exact correspondence between the pseudo-code and the Java program. Often a single line of pseudo-code corresponds to several parts of the program. The pseudo-code, however, should accurately convey the algorithm that will be used by the program.

The program itself starts by obtaining the width and height of the window upon which the bugs appear. It uses two class methods that were documented in the *PaintBug*'s interface in Section 6.2.5. It then declares variables that are used throughout the program.

Lastly, it creates the two *PaintBug* objects. The creation of the *PaintBug* objects use a constructor that is part of the interface of the *PaintBug* class. The particular constructor chosen allows the program to specify the beginning location of the *PaintBug* object.

The first bug is created about 1/3 of the way from the left side of the window. The second bug is placed 1/3 of the way from the right side of the window.

Execution now enters the `do` loop. A `do` loop is used because at least one (partial) iteration of the loop always occurs and the calculation of the distance between the two bugs most naturally occurs in the loop.

At the beginning of the loop, the two bugs are set to random directions. The first bug is then moved and `x1` and `y1` are set to its current location. The `if` statement contains a Boolean expression that evaluates to `true` if the bug's location is off the window. Note that this is the opposite of the expression in the `while` statement in

the *RandomWalk* program that evaluates to **true** only if the bug's location is on the window.

If the expression evaluates to **true** (if the bug is off the window), then the **break** statement is executed and execution leaves the loop. Otherwise, the second bug is moved, *x2* and *y2* are set to its current location, and the Boolean expression in the second **if** statement is evaluated to determine if the second bug is off the window. If it is (the expression evaluates to **true**), then the second **break** statement is executed and execution leaves the loop.

The last statement in the loop calculates the distance between the two bugs using the Pythagorean Theorem.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

The loop continues execution if the Boolean expression in the **while** statement evaluates to **true**, i.e. if the distance between the two bugs is greater than 50 pixels. If the distance is less than or equal to 50, then expression evaluates to **false** and execution leaves the loop.

8.6.4 Break vs. Loop Restructuring

While the use of the **break** statement usually allows for Java code to more closely resemble the programmer's original algorithm, it does mean that execution can jump from the middle of a loop to the end of the loop. This can make programs harder to follow, as it is no longer possible to instantly determine the exit conditions of a loop simply by looking at the top or bottom of the loop. This text takes a middle road. In most of the examples, we restructure programs so that they do not exit from the middle of loops. However, in cases where doing so would substantially add to the complexity of the program, we use the **break** statement.

8.7 Nesting Loops

The loop statements in Java can contain assignment statements, **if** statements, calls to methods, and even other loops. Placing a loop inside another loop is called **nesting** loops.

In a nested loop, there is an outer loop and an inner loop. In each iteration of the outer loop, the inner loop is executed, likely running the loop statements of the inner loop multiple times.

Example of Nested Loops

For example, the statements

```
for (int i = 1 ; i <= 3 ; i++)
{
    for (int j = 10 ; j <= 12 ; j++)
    {
        System.out.println (i + "," + j);
    } // for (j)
} // for (i)
```

produce the output

```
1 10
1 11
1 12
2 10
2 11
2 12
3 10
3 11
3 12
```

Each time through the outer loop, the inner loop performs 3 iterations with j ranging from 10 to 12.

Another Example of Nested Loops

Here is the pseudo-code for a times-table program.

- output column header
- for row: 1 .. 10
 - output row header (no newline)
 - for columns: 1 .. 10
 - output row * column (no newline)
 - output newline

For each iteration of the outer loop (for row: 1..10), the program outputs a row header and then executes the inner loop (for columns: 1..10). The inner loop executes 10 times, outputting the results for the appropriate line in the times table. Finally, the inner loop terminates and execution continues in the outer loop, causing it to output the newline.

Here is part of the program that implements the pseudo-code.

```
// From main method of the "TimesTable" class.
Stdout.println (" X   1   2   3   4   5   6   7   8   9  10");
Stdout.println (" + --- --- --- --- --- --- --- --- ---");
for (int row = 1 ; row <= 10 ; row++)
{
    Stdout.print (row, 2);      // Row header
    Stdout.print ("|");
    for (int column = 1 ; column <= 10 ; column++)
    {
```

```

        Stdout.print (row * column, 4);
    } // for (column)
    Stdout.println ();
} // for (row)

```

Here is the output from the *TimesTable* program.

```

X   1  2  3  4  5  6  7  8  9 10
+ ---
1 |  1  2  3  4  5  6  7  8  9 10
2 |  2  4  6  8 10 12 14 16 18 20
3 |  3  6  9 12 15 18 21 24 27 30
4 |  4  8 12 16 20 24 28 32 36 40
5 |  5 10 15 20 25 30 35 40 45 50
6 |  6 12 18 24 30 36 42 48 54 60
7 |  7 14 21 28 35 42 49 56 63 70
8 |  8 16 24 32 40 48 56 64 72 80
9 |  9 18 27 36 45 54 63 72 81 90
10| 10 20 30 40 50 60 70 80 90 100

```

Here is the trace of the execution of the `for` loop. The line being executed is indicated with an asterisk (*). Spaces in the output are denoted by ^ symbols.

<pre> // The "TimesTable" class. * Stdout.println (" X 1 2 3 4 ... Stdout.println (" + --- --- --- --- ... for (int row = 1 ; row <= 10 ; row++) { Stdout.print (row, 2); Stdout.print (" "); for (int col = 1 ; col <= 10 ; col++) { Stdout.print (row * col, 4); } // for (column) Stdout.println (); } // for (row) </pre>	Output the first line of the column header.	
<pre> Stdout.println (" X 1 2 3 4 ... * Stdout.println (" + --- --- --- --- ... for (int row = 1 ; row <= 10 ; row++) </pre>	Output the second line of the column header.	
<pre> * for (int row = 1 ; row <= 10 ; row++) { Stdout.print (row, 2); Stdout.print (" "); for (int col = 1 ; col <= 10 ; col++) { Stdout.print (row * col, 4); } // for (column) Stdout.println (); } // for (row) </pre>	The index variable (<i>row</i>) is declared and assigned the initial value (1). The comparison (<i>row <= 10</i>) evaluates to true . The loop statements of the outer loop are executed.	<i>row</i> =1
<pre> for (int row = 1 ; row <= 10 ; row++) { * Stdout.print (row, 2); Stdout.print (" "); </pre>	Output “^1” on the system console.	<i>row</i> =1

<pre> for (int row = 1 ; row <= 10 ; row++) { Stdout.print (row, 2); * Stdout.print (" "); </pre>	Output “ ” on the system console.	
<pre> for (int row = 1 ; row <= 10 ; row++) { Stdout.print (row, 2); Stdout.print (" "); * for (int col = 1 ; col <= 10 ; col++) { Stdout.print (row * col, 4); } // for (column) Stdout.println (); } // for (row) </pre>	The index variable (<i>col</i>) is declared and assigned the initial value (1). The comparison (<i>col</i> <= 10) evaluates to true . The loop statements of the inner loop are executed.	<i>col</i> =1
<pre> for (int col = 1 ; col <= 10 ; col++) { * Stdout.print (row * col, 4); } // for (column) </pre>	Output “^^^ ” on the system console.	<i>row</i> =1 <i>col</i> =1
<pre> * for (int col = 1 ; col <= 10 ; col++) { Stdout.print (row * col, 4); } // for (column) </pre>	The index variable <i>col</i> is incremented to 2. The comparison evaluates to true . The loop statements are executed.	<i>col</i> =2
<pre> for (int col = 1 ; col <= 10 ; col++) { * Stdout.print (row * col, 4); } // for (column) </pre>	Output “^^^2” on the system console.	<i>row</i> =1 <i>col</i> =2
<pre> * for (int col = 1 ; col <= 10 ; col++) { Stdout.print (row * col, 4); } // for (column) </pre>	The index variable <i>col</i> is incremented to 3. The comparison evaluates to true . The loop statements are executed.	<i>col</i> =3
<pre> for (int col = 1 ; col <= 10 ; col++) { * Stdout.print (row * col, 4); } // for (column) </pre>	Output “^^^3” on the system console.	<i>row</i> =1 <i>col</i> =3
The inner for loop continues to execute until <i>col</i> is 10.		<i>col</i> =10
<pre> for (int col = 1 ; col <= 10 ; col++) { * Stdout.print (row * col, 4); } // for (column) </pre>	Output “^^^10” on the system console.	<i>row</i> =1 <i>col</i> =10

<pre> for (int row = 1 ; row <= 10 ; row++) { Stdout.print (row, 2); Stdout.print (" "); * for (int col = 1 ; col <= 10 ; col++) { Stdout.print (row * col, 4); } // for (column) Stdout.println (); } // for (row) </pre>	<p>The index variable <i>col</i> is incremented to 11. The comparison evaluates to false. Execution continues following the loop statements.</p>	<p><i>col</i>=11</p>
<pre> for (int row = 1 ; row <= 10 ; row++) { Stdout.print (row, 2); Stdout.print (" "); for (int col = 1 ; col <= 10 ; col++) { Stdout.print (row * col, 4); } // for (column) * Stdout.println (); } // for (row) </pre>	<p>Output a newline on the system console. The one-times table line is completed</p>	
<pre> * for (int row = 1 ; row <= 10 ; row++) { Stdout.print (row, 2); Stdout.print (" "); for (int col = 1 ; col <= 10 ; col++) { Stdout.print (row * col, 4); } // for (column) Stdout.println (); } // for (row) </pre>	<p>The index variable <i>row</i> is incremented to 2. The comparison evaluates to true. The loop statements are executed.</p>	
<p>The outer for loop continues to execute until <i>row</i> is 10. Each time through the outer loop the inner loop executes 10 times to produce a line in the times table.</p>		

Important Points About Nested Loops

There are some important points to know about nested loops.

- A loop within another loop must be completely contained by the exterior loop. Loops cannot partially “overlap”.
- The inner and outer loops are not related. They do not have to be the same type of loop.
- When nesting similar loops, use a comment on the closing brace of each loop to make it clear which loop is being closed. Often the index variable of a **for** loop or the Boolean expression of a **while** loop is used to differentiate the closing braces of different loops.
- A **break** statement with the inner loop statements only breaks out of the innermost loop. A **break** statement in the outer loop statements breaks out of the outermost loop.

Another Loop Example

Here is a variant of the *RandomWalk* class called *MultipleRandomWalk* that draws a sequence of *PaintBug* objects doing a random walk. As each bug moves off the window, a new *PaintBug* object is created in the center of the screen with a different color.

```
// The "MultipleRandomWalk" class.
import hsa.PaintBug;
public class MultipleRandomWalk
{
    public static void main (String[] args)
    {
        while (true)
        {
            PaintBug bug = new PaintBug ();
            int width = PaintBug.getXSize ();
            int height = PaintBug.getYSize ();
            double x, y;

            do
            {
                double angle = Math.random () * 360;
                bug.setDirection (angle);
                bug.move (20);
                x = bug.getXPos ();
                y = bug.getYPos ();
            }
            while ((0 <= x) && (x < width) &&
                (0 <= y) && (y < height));
        } // while (true)
    } // main method
} // MultipleRandomWalk class
```

8.8 Technical Terms

conditional loop	loop
counted loop	loop statement
index variable	nesting
infinite loop	null statement
initial value	pseudo-code
iterates	sequential execution
iterates over	sentinel value
iterates through	termination value
iteration	

8.9 What You Should Know

- What a counted loop is. [8.1]
- What a conditional loop is. [8.1]
- What pseudo-code is. [8.2]
- How to write a `for` loop. [8.3.1]
- How to use a `for` loop to count up, count down, count by values other than one. [8.3.3]
- How to write a `while` loop. [8.4.1]
- How to write a `do` loop. [8.4.2]
- How to choose between the different types of loops. [8.5]
- How to break out of a loop. [8.6]
- How to nest loops within one another. [8.7]
- The definitions of all the technical terms. [General]

8.10 Quiz Answers

8.10.1 Answers – Simple Counted Loops (pg. 288)

Here are the worked out answers. Work out the solutions yourself before looking at these answers.

The initialization or increment and the comparison that takes place in the `for` statement is on the left. In each solution, the output is on the right.

a. Question

```
for (int j = 0 ; j < 4 ; j++)
{
    System.out.print ("Hi!");
} // for
```

Answer:

```
(j = 0, j < 4 is true, Loop continues)      Hi!
(j = 1, j < 4 is true, Loop continues)      Hi!
(j = 2, j < 4 is true, Loop continues)      Hi!
(j = 3, j < 4 is true, Loop continues)      Hi!
(j = 4, j < 4 is false, Loop exits)
```

b. Question:

```
for (int j = 0 ; j < 4 ; j++)
{
    System.out.println (5 - j);
} // for
```

Answer:

```
(j = 0, j < 4 is true, Loop continues)      5
(j = 1, j < 4 is true, Loop continues)      4
(j = 2, j < 4 is true, Loop continues)      3
(j = 3, j < 4 is true, Loop continues)      2
(j = 4, j < 4 is false, Loop exits)
```

c. Question:

```
for (int j = 1000 ; j <= 1005 ; j++)
{
    System.out.println (5 - j);
} // for
```

Answer:

```
(j = 1000, j <= 1005 is true, Loop continues)      -995
(j = 1001, j <= 1005 is true, Loop continues)      -996
(j = 1002, j <= 1005 is true, Loop continues)      -997
(j = 1003, j <= 1005 is true, Loop continues)      -998
(j = 1004, j <= 1005 is true, Loop continues)      -999
(j = 1005, j <= 1005 is true, Loop continues)      -1000
(j = 1006, j <= 1005 is false, Loop exits)
```

d. Question:

```
for (int j = 10 ; j < 12 ; j++);
{
    System.out.println ("Hello");
} // for
```

Answer:

The **for** statement has a semicolon after it. This means that the loop statements are comprised of a single null statement

```
(j = 10, j < 12 is true, Loop continues)      [Nothing]
(j = 11, j < 12 is true, Loop continues)      [Nothing]
(j = 12, j < 12 is false, Loop exits)
```

The statements in curly braces following the loop construct is now executed.

Hello

e. Question:

```
for (int j = 6 ; j < 6 ; j++)
{
    System.out.println (j);
} // for
```

Answer:

*(j = 6, j < 6 is **false**, Loop exits)* [No output]

There is no output to this program. The initial check of the termination condition fails so the loop statements are never executed.

f. Question:

```
for (int j = 10 ; j <= 10 ; j++)
{
    System.out.println (j);
} // for
```

Answer:

*(j = 10, j <= 10 is **true**, Loop continues)* 10
*(j = 11, j <= 10 is **false**, Loop exits)*

g. Question:

```
for (int j = -6 ; j < -4 ; j++)
{
    System.out.println (5 - j);
} // for
```

Answer:

*(j = -6, j < -4 is **true**, Loop continues)* 11
*(j = -5, j < -4 is **true**, Loop continues)* 10
*(j = -4, j < -4 is **false**, Loop exits)*

8.10.2 Answers – Counted Loops (pg. 292)

Here are the worked out answers. Work out the solutions yourself before looking at these answers.

In each solution, the output is on the left. The initialization or increment and the comparison that takes place in the `for` statement is on the right.

a. Question:

```
for (int j = 6 ; j > 2 ; j--)
{
    System.out.print ("Hi!");
} // for
```

Answer:

*(j = 6, j > 2 is **true**, Loop continues)* Hi!
*(j = 5, j > 2 is **true**, Loop continues)* Hi!
*(j = 4, j > 2 is **true**, Loop continues)* Hi!
*(j = 3, j > 2 is **true**, Loop continues)* Hi!
*(j = 2, j > 2 is **false**, Loop exits)*

b. Question:

```
for (int j = 0 ; j < 4 ; j += 2)
{
    System.out.println (5 - j);
} // for
```

Answer:

```
(j = 0, j < 4 is true, Loop continues)      5
(j = 2, j < 4 is true, Loop continues)      3
(j = 4, j < 4 is false, Loop exits)
```

c. Question:

```
for (int j = 12 ; j <= 20 ; j--)
{
    System.out.println (j);
} // for
```

Answer:

```
(j = 12, j <= 20 is true, Loop continues)    12
(j = 11, j <= 20 is true, Loop continues)    11
(j = 10, j <= 20 is true, Loop continues)    10
(j = 9, j <= 20 is true, Loop continues)     9
(j = 8, j <= 20 is true, Loop continues)     8
...                                             ...
```

This program is in an infinite loop. The index variable decreases each time through the loop. The comparison will always be true.

d. Question:

```
for (int j = -5 ; j < 5 ; j++);
{
    System.out.println (j);
} // for
```

Answer:

```
(j = -5, j < 5 is true, Loop continues)      -5
(j = -4, j < 5 is true, Loop continues)      -4
(j = -3, j < 5 is true, Loop continues)      -3
(j = -2, j < 5 is true, Loop continues)      -2
(j = -1, j < 5 is true, Loop continues)      -1
(j = 0, j < 5 is true, Loop continues)       0
(j = 1, j < 5 is true, Loop continues)       1
(j = 2, j < 5 is true, Loop continues)       2
(j = 3, j < 5 is true, Loop continues)       3
(j = 4, j < 5 is true, Loop continues)       4
(j = 5, j < 5 is false, Loop exits)
```

e. Question:

```
for (int j = -3 ; j >= -21 ; j -= 3)
{
    System.out.println (j);
} // for
```

Answer:

```
(j = -3, j >= -21 is true, Loop continues)      -3
(j = -6, j >= -21 is true, Loop continues)      -6
(j = -9, j >= -21 is true, Loop continues)      -9
(j = -12, j >= -21 is true, Loop continues)     -12
(j = -15, j >= -21 is true, Loop continues)     -15
(j = -18, j >= -21 is true, Loop continues)     -18
(j = -21, j >= -21 is true, Loop continues)     -21
(j = -24, j >= -21 is false, Loop exits)
```

f. Question:

```
for (int j = 10 ; j <= 3 ; j--)
{
    System.out.println (j - 5);
} // for
```

Answer:

```
(j = 10, j <= 3 is false, Loop exits)
```

There is no output to this program. The initial check of the termination condition fails so the loop statements are never executed.

g. Question:

```
for (int j = -10 ; j < 10 ; j += 30)
{
    System.out.println (j);
} // for
```

Answer:

```
(j = -10, j < 10 is true, Loop continues)      -10
(j = 20, j < 10 is false, Loop exits)
```

8.10.3 Answers – While Loops (pg. 295)

Here are the worked out answers. Work out the solutions yourself before looking at these answers.

In each solution, the output is on the left. The initialization or increment and the comparison that takes place in the `for` statement is on the right.

a. Question:

```
int j = 1
while (j <= 8)
{
    System.out.print (j);
    j = j * 2;
} // while
```

Answer:

```
(j = 1, j <= 8 is true, Loop continues)      1
(j = 2, j <= 8 is true, Loop continues)      2
(j = 4, j <= 8 is true, Loop continues)      4
(j = 8, j <= 8 is true, Loop continues)      8
(j = 16, j <= 8 is false, Loop exits)
```

b. Question:

```
int j = 20
while (j > 3)
{
    System.out.println (j);
    j = j / 2;
} // while
```

Answer:

```
(j = 20, j > 3 is true, Loop continues)      20
(j = 10, j > 3 is true, Loop continues)      10
(j = 5, j > 3 is true, Loop continues)      5
(j = 2, j > 3 is false, Loop exits)
```

c. Question:

```
int number = 1, sum = 20;
while (number < 5);
{
    sum += number;
    number++;
    System.out.println (sum);
} // while
```

Answer:

```
(number = 1, number < 5 is true, Loop continues)
    sum becomes 21, number becomes 2      21
(number = 2, number < 5 is true, Loop continues)
    sum becomes 23, number becomes 3      23
(number = 3, number < 5 is true, Loop continues)
    sum becomes 26, number becomes 4      28
(number = 4, number < 5 is true, Loop continues)
    sum becomes 30, number becomes 5      30
(number = 5, number < 5 is false, Loop exits)
```

d. Question:

```
int number = 1, product = 20;
while (product < 100)
{
    product *= number;
    number++;
    System.out.println (number);
} // while
```

Answer:

```
(product = 20, product < 100 is true, Loop continues)
    product becomes 20, number becomes 2      2
(product = 20, product < 100 is true, Loop continues)
    product becomes 40, number becomes 3      3
(product = 40, product < 100 is true, Loop continues)
    product becomes 120, number becomes 4     4
(product = 120, product < 100 is false, Loop exits)
```

e. Question:

```
int number = 3;
while (number < 20)
{
    number = number + (number - 2);
    System.out.println (number);
} // while
```

Answer:

```
(number = 3, number < 20 is true, Loop continues)
    number becomes 4      4
(number = 4, number < 20 is true, Loop continues)
    number becomes 6      6
(number = 6, number < 20 is true, Loop continues)
    number becomes 10     10
(number = 10, number < 20 is true, Loop continues)
    number becomes 18     18
(number = 18, number < 20 is true, Loop continues)
    number becomes 34     34
(number = 34, number < 20 is false, Loop exits)
```

f. Question:

```
int number = 5
while (Math.pow (number, 2) < 50)
{
    System.out.println (number);
    number++;
} // while
```

Answer:

```
(number = 5, number2 < 50 is true, Loop continues) 5
(number = 6, number2 < 50 is true, Loop continues) 6
(number = 7, number2 < 50 is true, Loop continues) 7
(number = 8, number2 < 50 is false, Loop exits)
```

g. Question:

```
int j = -10
while (!(j > 10))
{
    System.out.println (j);
    j += 3;
} // while
```

Answer:

```
(j = -10, !(j > 10) is true, Loop continues) -10
(j = -7, !(j > 10) is true, Loop continues) -7
(j = -4, !(j > 10) is true, Loop continues) -4
(j = -1, !(j > 10) is true, Loop continues) -1
(j = 2, !(j > 10) is true, Loop continues) 2
(j = 5, !(j > 10) is true, Loop continues) 5
(j = 8, !(j > 10) is true, Loop continues) 8
(j = 11, !(j > 10) is false, Loop exits)
```

8.11 Exercises

1. Create a program called *CubeRoots* that outputs a table of numbers and their cube roots from 10 to 50. The table should line up and the cube roots should be output to 4 decimal places. [8.3.1]
2. Create a program called *SumSequence* that obtains a number from the user and then outputs the sum of 1 through the number ($1 + 2 + 3 + \dots + \text{number}$) using a **for** loop. [8.3.1]
3. Create a program called *Factorial* that obtains a number from the user and then outputs the product of 1 through the number ($1 * 2 * 3 * \dots * \text{number}$) using a **for** loop. What happens if the result is larger than can be held in an **int**. [8.3.1]
4. Create a program called *Fibonacci* that obtains a number n from the user and then outputs the first n numbers in the Fibonacci numbers. The Fibonacci sequence starts with 1 and 1. Each additional element in the sequence is derived by adding the previous two elements in the sequence. Here are the first 10 numbers. [8.3.1]

1 1 2 3 5 8 13 21 34 55

5. Create a program called *Triangle* that creates a triangle made of asterisks using a loop. [8.3.1]

```
*
**
***
****
*****
*****
```

6. Create a program called *Diamond* that obtains creates a diamond made of asterisks using two loops. [8.3.1]

```

*
 ***
  *****
 *****
*****
 *****
  *****
   ***
    *
```

7. Create a program called *BetterBox* that uses the *hsa.PaintBug* class. The program should draw a box using a `for` loop. [8.3.1]
8. Create a program called *SpiralLoop* that uses the *hsa.PaintBug* class. The program should draw a square spiral using a `for` loop. Each edge it draws should be slightly shorter than the one before it. [8.3.1]
9. Create a program called *ManyV* that uses the *hsa.PaintBug* class. The program starts a *PaintBug* object near the upper-left corner of the window and draws 5 large 'V's, connected to each other, making a back and forth pattern. [8.3.1]
10. Create a program called *SquareTable* that outputs numbers and their squares from 1 to 80 in a table as follows. [8.3.1]

Num	Square	Num	Square	Num	Square	Num	Square
1	1	21	441	41	1681	61	3721
2	4	22	484	42	1764	62	3864
...							
20	400	40	1600	60	3600	80	6400

11. Create a program called *By3* that outputs multiples of 3s from 30 to 60 using a `for` loop. [8.3.3]
12. Create a program called *PopulationGrowth* that calculates how many years it will be until the population of a city exceeds 1,000,000 people. The current population of the city is 150,000 and it is growing at 5% a year. [8.4]

13. Create a program called *CountDown* that obtains a number from the user and counts backwards from 100 by 5s, outputting “I stopped.” when the count would be less than the number specified by the user. The numbers should line up. [8.4]

```
What number do I stop at? 82
100
 95
 90
 85
I stopped.
```

14. Create a program called *Point* that simulates the playing of a simple game. Roll a single die to get a value from 1 to 6. This is called your *point*. Now keep rolling until you roll the same value again (your point). Output what the point was, the list of rolls that were made until the point was rolled, and a count of how many times the die was rolled until the point was rolled. [8.4]
15. Create a program called *SumDigits* that obtains an integer from the user and then sums the digits. The program should use the remainder and integer division operators. [8.4]
16. Create a program called *CalendarMonth* that obtains the day of the week that a month starts in and the number of days in the month. It then outputs a well-formatted calendar for the month. [8.4]

```
Day the month starts on (1 for Sunday): 3
Number of days in month: 30
```

```
Sun Mon Tue Wed Thu Fri Sat
      1  2  3  4  5
 6   7   8   9  10  11  12
13  14  15  16  17  18  19
20  21  22  23  24  25  26
27  28  29  30
```

17. Create a program called *Factors* that obtains an integer from the user and outputs all of its divisors on one line. [8.4]

```
Enter a number: 12
1 2 3 4 6 12
```

18. Create a program called *PerfectNumbers* that outputs the first three perfect numbers. A perfect number is a number that is equal to the sum of all of its divisors (excluding itself). For example 6 is a perfect number because $6 = 1 + 2 + 3$. [8.4]
19. Create a program called *simpleCalc* that reads in a simple expression in the form

```
[number] [operator] [number]
```

and outputs the result. The program should halt if both numbers are 0, otherwise it should output the result. The numbers are floating-point (use the *Stdin.readDouble* method to read them). The operator is either +, -, * or / (use

`Stdin.readChar` to read it). The program does not need to handle badly formed expressions. [8.6]

```
Enter expression: 5.5 * 8
44
Enter expression: 12 / 8
1.5
Enter expression: 0 + 0
```

20. Write the output of the following program without entering the program and executing it. Note all spaces and newlines carefully. [8.7]

```
// The "NestedLoops1" class.
public class NestedLoops
{
    public static void main (String[] args)
    {
        for (int i = 5 ; i < 10 ; i++)
        {
            for (int j = i / 2 ; j < i + 3 ; j++)
            {
                System.out.print (i + " " + j + " ");
            } // for
            System.out.println ();
        } // for
    } // main method
} // NestedLoops1 class
```

21. Create a program called *AllFactors* that outputs the numbers from 1 to 50 along with their divisors. Each number and its divisors should be on the same line. [8.7]
22. Create a program called *ManyFactors* that obtains two numbers from the user and outputs the number between the two numbers that has the most divisors along with the number of divisors. [8.7]

```
Enter lower bound: 10
Enter upper bound: 20
12 has the most divisors: 6
```

